Toward a modularization of Pharo: Analysis of the design space for a new module system.

Camille Teruel, Stéphane Ducasse, Marcus Denker

MajecSTIC 2012
Lille, France, du 29 au 31 octobre 2012

Toward a modularization of Pharo : Analysis of the design space for a new module system.

Camille Teruel[1], Stéphane Ducasse[2], Marcus Denker[3]

Université Lille 1, LIFL, équipe RMod - France.

1 :camille.teruel@gmail.com, 2 :stephane.ducasse@inria.fr, 3 :marcus.denker@inria.fr

Résumé
Pharo est un langage de programmation orienté objet et réflexif descendant de Smalltalk. Actuellement, il ne propose pas de notion de module ou d'espace de noms. Toutes les classes et les variables globales sont visibles depuis l'ensemble du système ; de nombreuses dépendences peuvent donc apparaître et rendre le système monolithique. Pharo doit donc fournir un nouveau concept pour permettre sa modularisation. Transformer Pharo en système modulaire n'est pas une tâche facile : pour permettre la migration, ce nouveau concept doit supporter les extensions locales de classes et les dépendances circulaires. De plus, l'espace des solutions est large : non seulement le modéle doit définir la sémantique des extensions locales de classes, mais il doit également faire de même pour la visibilité et pour la paramétrisation des modules. Dans ce contexte, cet article présente une analyse de l'espace de conception pour un système de module au travers de trois axes de conception : les extensions locales de classes, la résolution des dépendances et la paramétrisation.

Abstract
Smalltalk is a reflective object-oriented programming language. Over the years, it has influenced many other programming languages and evolved into many variants. However, it does not offer the notion of namespace or module. Because all classes and global variables are visible from the whole system, numerous dependencies may be introduced and lead to a monolithic system. As a descendant of Smalltalk, Pharo should provide a language construct conceived with modularity in mind. Besides, transforming Pharo into a modular system it not an easy task : in order to allow the migration, the modular construct must support local class extensions and circular dependencies. Moreover, the design space is large : not only do this construct have to define the semantics of local class extensions and dependency resolution, but it should also do the same with visibility and module parametrization. In this context, this article presents an analysis of the design space through three design axis : local class extensions, dependency resolution and module parametrization.

Keywords: object oriented programming, Pharo, modularization, local class extensions, dependency resolution

1. Introduction

Modularity is considered to be a desirable feature for a software system. It aims to improve reuse, maintenance and independent development, making development more efficient. The main goal is to be able to build new software systems by combining reusable subsystems together. Most programming languages come with some built-in modular construct that aims to support modularity at the language level. These constructs bear miscellaneous names and support various sets of features accordingly to their own concerns. Because most of these names are widely used in computer sciences and software engineering (like *component* and *package* for example), we will refer to these constructs via the generic term *module*.

1.1. Definition

All successful programming languages need to cope with huge code bases at some point. Therefore, all modules are born because of the same need : to put related things together. The first concern of a module is thus to bundle correlated programming entities into a new one. The nature of these entities depends mostly on the language paradigm : in functional languages like OCaml these entities are usually *types* and *values* [11] whereas in object-oriented languages they are mostly *classes* or *objects*. The following definition of *module* conforms to this concern :

> A module is a capsule containing (definitions of) items. The module draws a strong (syntactic) boundary between items defined inside it and items defined outside in other modules [16].

Modules should also be concerned with visibility in order to control the access of the entities they contain. This feature is called *information hiding*. The granularity and the modalities of the control can vary from one module concept to another. In most languages, information hiding is limited to a set of fixed policies [1]. Other modules can provide more sophisticated control means. For example, a single module can expose different views of itself [2], helping the developer to produce a clean *interface segregation* that leads to narrow role-oriented APIs. Even more advanced mechanism that involves users, groups and permissions may be conceivable. This second concern is also found in another definition of module :

> Modules are program units that manage the visibility and accessibility of names. A module defines a set of constant bindings between names and objects [18].

Information hiding and bundling of related entities are orthogonal concerns. This report discuss the latter. But this concern implies another one : dependency management. Indeed, even if it is possible to design self-sufficient sets of related entities, in practice most of them have dependencies. Consequently, we state that a module is a language construct that at least permits to bundle, control the visibility and manage the dependencies of a set of related software entities.
In addition, a modular construct can provide additional features. For example, some module have been seen acting like mixins [12] (as in Ruby), others provide dynamic linking mechanisms [9] (as in Racket) while others got unified with objects [7]. Even more specific features have been proposed like support for late bound classes [15] and class extensions [4,17]. Since class extensions are widely used in Pharo, this latter feature will be discussed in details later.

2. Vocabulary

In this section, we define the vocabulary used in the rest of the paper to avoid confusion.
- A *binding* is the association of a name and a value.
- An *environment* is a structure that maps names to values, it consists of a list of *bindings*. The term *namespace* is also common.
- A *module* is a programming unit that provides an environment for a set of immutable objects. These objects can either be classes or base objects that represent constants. The classes defined by a module may contain methods that refer to names that are not bound in the enclosing module. These names are called *dependencies*.
- A *configuration* for a given module is a structure that maps its dependencies to object defined in other modules.
- A module together with a suitable configuration is called a *concrete module*.

Making a clear separation between the concepts of *module*, *configuration* and *concrete module* is mandatory to support generic programming. Indeed, it permits to parametrize a module with classes and base objects to give a new concrete module.

3. Context

Smalltalk is a fully reflective object-oriented language and platform. It has been inspiring a lot of object oriented languages over the years and has evolved into many variants. Pharo is one of

1. such as the *public*, *protected* and *private* keywords for Java packages
2. An example of this feature can be found in Jigsaw, Java's future module system

these variants forked from another Smalltalk implementation called Squeak. Pharo inherits from the qualities from its ancestor, but also from their flaws : it has no notion of namespace nor of class encapsulation. After a quick glance at Smalltalk, we describe the current Pharo problems and presenting class extensions. Finally, we state a set of requirements that a module construct for Pharo must support.

3.1. A small Smalltalk overview

Smalltalk is an object-oriented programming language and platform that have the following properties :

- *Reflection :* Smalltalk is said reflective because it reifies its own meta-model, a feature that permits to extend the language easily.
- *Purity :* Smalltalk is a pure object-oriented language : everything is represented as an object, even primitive types and classes.
- *Image-based :* Contrary to most programming language, Smalltalk does not rely on an underlying file structure to store the code. Instead, it uses an *image i.e.,* a full snapshot of all the objects that exist in a given system. An image is thus an alive system that one can modify incrementally in order to develop a new software.
- *Late binding :* Like most dynamic object-oriented languages, Smalltalk uses message passing and late-binding (sometimes called duck-typing).

The syntax for message passing differs from the more famous Java one. The next example shows the differences.

```
bob send: letter to: alice.
bob.sendTo(letter,alice);
```

3.2. Problematic

The unique environment problem

In Pharo, all classes and globals are referenced via a unique global environment (*i.e.,* a namespace), an instance of the class SystemDictionary. This environment is accessed via Smalltalk globals (see Figure 1). Note that each class has a back-pointer to the system dictionary. This pointer is a good extension point to introduce environments. Since a single name can refer to different objects in different software projects, name clashes can appear when one tries to integrate independently developed projects. In Pharo, the practice is to prefix class names with acronyms acting as poor's man namespaces. While such name clashes look problematic, relying on environments only defer the problem to module names. Indeed, programmers still have to pay attention that module names do not clash.

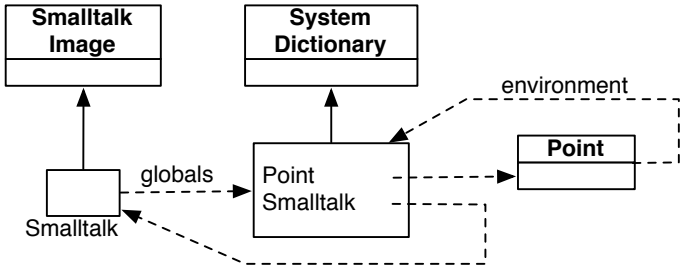

FIGURE 1 – Pharo single namespace named Smalltalk globals, instance of the class SystemDictionary.

Introducing environments in the form of modules is needed not only to manage name clashes, but more importantly to provide an abstraction supporting encapsulation for a set of related classes. This will help identifying dependencies and will serve as a basis for providing the mandatory information hiding principles.

Class Extensions

Smalltalk makes heavy use of a feature called *class extensions*. A class extension consists in adding methods to a class that already exists in the system. Similar features in other dynamic languages are called *Monkey Patching* or *Open Classes*. In Figure 2, when the package Network is loaded, it adds a new method asUrl to the class String. When the package Network is unloaded, the asUrl method is then removed from the class String.

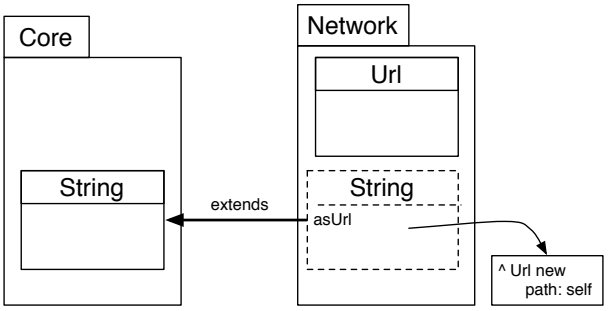

FIGURE 2 – Class extension and Package in Smalltalk. The class String is extended with a class extension asUrl.

Class extension is a powerful mechanism because a class can be extended without subclassing it. Whereas subclassing requires that instances of the new classes are used in place of the previous ones, the same instances can be used with class extensions. However, because the effect of a class extension is global *i.e.*, visible from everywhere in the system, this concept brings several issues :
– The method dictionaries of extended classes get polluted. A client can see methods that is not supposed to use.
– Conflicts can appear when two independent developed modules add the same method to a class. In this case, the last loaded module destroys the class extension from the other module. Thus loading a module can break other independent modules.
– These new methods usually depend on external classes. This means that new dependencies are added in the extended classes. Moreover, most of the time these dependencies are located in the same module than the class extension and a circle appears in dependencies.

The introduction of a module concept is the occasion to revisit the scope and semantics of class extensions to make them local to a module. There are two choices to express the locality of a class extension : a classboxes semantic [4, 5] or a selector namespaces one [17]. Both solutions have advantages and drawbacks as we will discuss later.

Migration constraints

Because Pharo had been built within a single namespace for years, a lot of dependencies appeared and led to a monolithic system. Transforming a monolithic system into a modular one is not an easy task because a lot of circular dependencies exist. Therefore, we need to have a module concept that accepts circular dependencies in order to support the migration. Obviously, having circular dependencies in a system is not a good idea, but these dependencies already exist and we must handle them, at least temporarily.

3.3. Requirements

So far, we have identified a few requirements a module construct for Pharo must fulfill :
– A module is a construct that bundle related classes and objects together in a private environment.
– A module must be able to embed extensions of external classes. These extensions must be visible from this module only.
– Circular dependencies between modules must be supported.

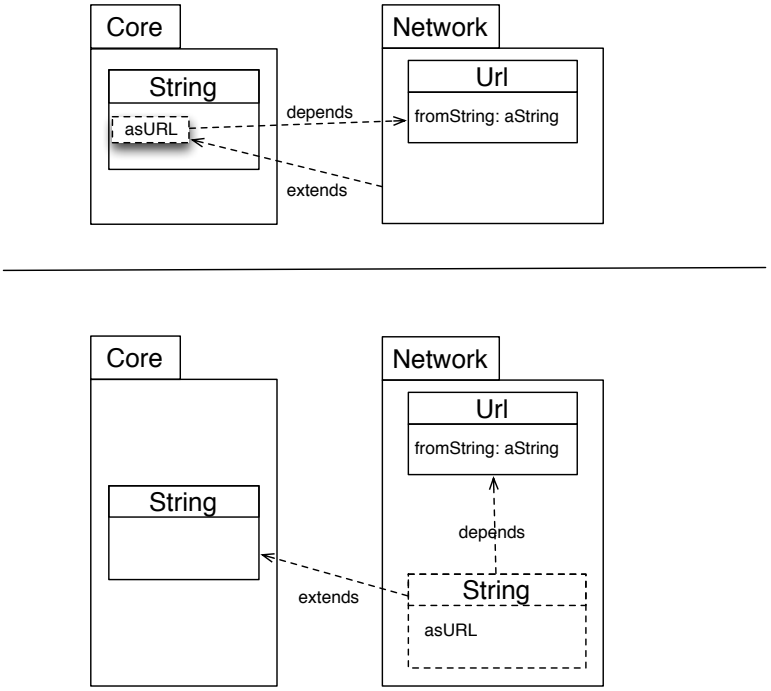

FIGURE 3 – Dependencies with global (top) and local (bottom) class extensions

4. Design Space : Class extensions

4.1. Rationales for local class extensions

A class extension permits the programmer to add new methods to a class that already exists in the system. The problem is that a class extension is globally visible : it modifies directly the method dictionary of the extended class. The consequence is that a lot of circular dependencies appear as explained in the Figure 3. This figure represents the dependencies introduced by global (top) and local (bottom) class extensions. A module Network defines a class URL and adds a new method to the class String defined in the module Core. In the two cases the module Network depends on the module Core since it extends its class String. The problem with global class extensions is that the class String depends on URL and a circular dependency between Core and Network appears. In the second case the visibility of the class extension of String is limited to Network and the circular dependency disappears. It is an application of the *Dependency Inversion Principle*. Note that local class extensions bring other benefits :

– They support both method addition and method redefinition.
– They avoid conflicts between different class extensions on a same method name.
– Because a module cannot bring changes outside itself, this solution respects the *Open/Closed Principle* [13] which states that software entities must allow extensions, but forbid modifications.

4.2. Context-sensitive vs context-free execution

Local class extensions can modify the behavior defined by a class locally. The question is to know which instances are affected by this modification. The answer to this question determines what kind of locality is intended :

– In a module, a class extension can affect every instance created within this module. These instances keep the same behavior all their life, even if they get referenced from another module. This means that the behavior of an instance is static and depends on where the instance is created. This kind of locality implies a context-free evaluation because the behavior of an object is determined statically.
– Otherwise, a class extension can affect every instance referenced within the corresponding mo-

dule. That means that instances of a class change their behavior dynamically according to the module that refers it. This kind of locality implies a context-sensitive evaluation.

The problem of context-free execution is that several instances of the same class referenced from a single module can have different behaviors. Because a module cannot know if an instance of a class it extends is affected by this extension or not, it cannot use this additional behavior and the extension becomes useless. Consequently, context-free execution is inconsistent with the intended use of class extensions. Therefore, we need context-sensitive execution that comes at the price of additional steps in the lookup algorithm. Context-sensitive execution unavoidably decreases performances.

4.3. Local rebinding

One advantage of local class extensions is their ability to locally overrides methods. The question is to determine when these methods are taken into account by the method lookup algorithm. If from the point of view of the module defining a local class extension, instances of the extended class behave as if the local class extension was global, the modular construct supports local rebinding [**?**, **?**]. This solution is really useful to handle unanticipated changes. It corresponds to the classboxes semantics. However, this semantics come at the cost of additional steps in the method lookup algorithm. These additional steps involve a performance loss and a modification of the virtual machine. Contrarily to classboxes, selector namespaces do not support local rebinding.

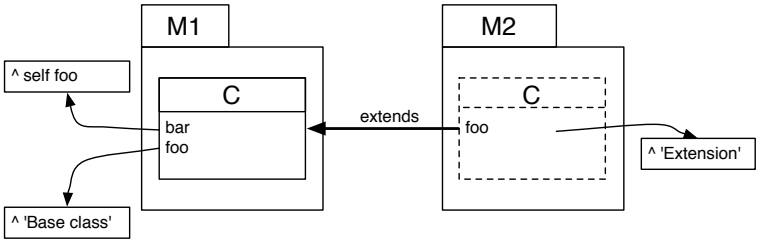

FIGURE 4 – Local rebinding : with local rebinding, C new foo evaluated in the context of M2 returns 'Extension'. Without local rebinding, the same expression returns 'Base class'.

The implementation of this mechanism can be done by prefixing every method's name with the name of the module that contains it. The method lookup takes the current module name into account to select the correct methods. All the prefixes are hidden from the developer. All methods are installed in the same dictionary but with different prefixes. This solution has the advantage to not slow down the execution and to not require virtual machine modification.

5. Design Space : Dependency resolution

Most constructs found in programming languages rely on explicit resolution hard-coded inside the construct (usually via *import statements*). However, it is not the only way to deal with dependency resolution. We present three variants : internal explicit resolution, external explicit resolution and lookup-based resolution. Then, we discuss the granularity of imports.

5.1. Explicit resolution

Explicit resolution is a trivial solution for managing dependencies. It can be either internal or external. Internal explicit resolution is found in most mainstream programming language via *import statements* (*e.g.*, Java packages). We call it internal because dependencies are expressed inside the construct definition. The problem of this solution is that it doesn't permit to reconfigure a given module without reconstructing it and so without loosing its identity. External explicit binding however permits to change the configuration of a module on-the-fly. This kind of dependency resolution is also called *dynamic linking* [9]. A similar idea is found in dependency injection frameworks where dependencies are injected into a class instead of letting a class make internal

imports. The consequence is that it is possible to change the dependencies without modifying the class. It is another application of the *Open/Close Principle* [13].

5.2. Lookup-based resolution

If the resolution is done via a lookup algorithm, that means that the topology of the module network is taken into account. This solution has two main flaws. First, you must find a solution with circles : ensure that a chain of module does not contain any circles or detect circle to stop the lookup. Secondly, the lookup can cross more than one module and so does dependencies. Therefore, the modification of a modules can have unexpected effects on indirectly related modules. For example, imagine a lookup chain made of three module : A -> B -> C. Lets say that a definition in A asks for the value associated with the name *foo*. The lookup begins in A which does not contains any value for the name foo. The lookup continue to B, which does not contain the binding so the lookup finally goes to C that returns the value of the binding it defines for the name *foo*. The consequence is that a modification of C can affect A whereas C is most probably not aware that A exists.

5.3. Granularity of resolution

For the granularity, imports can be enumerated explicitly or support wild-cards *i.e.,* the importation of all the definitions contained in a given module. While wild-cards look appealing for the users, they can lead to problems when an application evolves. For example, while an application may have no naming conflicts, if one module introduces a class with a conflicting name, the user will have to address the conflict even if he did not change his own code. In addition, explicit import allows one to understand the interface size between modules that is a valuable information for refactorisation.

6. Design Space : Module parametrization

On certain occasions, the programmer may want to reuse a module but be able to specify certain parameters such as the classes to be used given a scenario. For example, we can imagine that we have a module for the CollectionTest and that we would like to reuse this module to verify if an alternate library fulfills the same specification.

```
m2 := CollectionTest bind: #OrderedCollection to: MyCollection.
```

To support module parametrization, it is useful to make a clear distinction between an module and its configuration *i.e.,* a set of external bindings. A module together with a suitable configuration gives a concrete module that is usable. The problem with module parametrization arises when a module uses a class parameter as a superclass of a class defined in the module. Indeed, the virtual machine refers to instance variables by integers that correspond to their offset. The methods of the subclass may refer to its instance variables but their offsets are not known. It is thus mandatory to specialize these methods when the parameter is given to adapt their offsets. All classes that can be passed as a module parameter don't have the same number of instance variable. That means that the methods objects of an abstract subclass cannot be shared among all the different concrete modules. Consequently, each time an abstract module is parametrized with a class it uses as a superclass, all the methods that refers to the instance variable of the abstract subclass must be copied in the concrete module and recompiled to adapt their offsets. This mechanism is called *copy-down* and is already used for mixin applications [2].

7. Conclusion

In this article, we proposed an analysis of the design space for a module construct that respects Pharo's specific requirements. During this analysis, we gathered a set of requirements that this construct may fulfill :

- Local class extensions : the module construct must support local class extensions with method addition and method redefinition but without local rebinding.
- Dependency resolution : resolution of dependencies must be explicit and external to a module. Dependencies are fulfilled one by one : there is no wild-card for imports.

– Module parametrization : it must be possible to define abstract modules that can be parametrized with different configurations to give concrete modules.

However, this set of requirements can be implemented in various ways. Our further work will focus on different implementations to measure their performances and their consequences on Pharo's maintainability. The road to the modularization of a system that consists of more than 5000 classes is not easy and will take a lot of time. Once the modular construct is ready to be used, we will have to migrate all development tools (class browser, version control manager, package management system, debugger...).

Bibliography

1. Davide Ancona et Elena Zucca. True modules for Java-like languages. In J. L. Knudsen, editor, *ECOOP 2001*, number 2072 in LNCS, pages 354–380. Springer Verlag, 2001.
2. Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, et Urs Hölzle. Mixins in Strongtalk. In *ECOOP '02 Workshop on Inheritance*, juin 2002.
3. Alexandre Bergel, Stéphane Ducasse, et Oscar Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 11(10) :1613–1644, novembre 2005.
4. Alexandre Bergel, Stéphane Ducasse, et Oscar Nierstrasz. Classbox/J : Controlling the scope of change in Java. In *Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA, 2005. ACM Press.
5. Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, et Roel Wuyts. Classboxes : Controlling visibility of class extensions. *Journal of Computer Languages, Systems and Structures*, 31(3-4) :107–126, décembre 2005.
6. Gilad Bracha et William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, octobre 1990.
7. Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, et Eliot Miranda. Modules as objects in newspeak. In *ECOOP 2009*, LNCS. Springer, 2009.
8. John Corwin, David F. Bacon, David Grove, et Chet Murthy. MJ : a rational module system for Java and its applications. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 241–254. ACM Press, 2003.
9. Matthew Flatt et Matthias Felleisen. Units : Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.
10. Yuuji Ichisugi et Akira Tanaka. Difference-based modules : A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 sur *LNCS*, Malaga, Spain, juin 2002. Springer Verlag.
11. Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3) :269–303, 2000.
12. Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly, 2001.
13. Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.
14. Todd Millstein, Mark Reay, et Craig Chambers. Relaxed multijava : balancing extensibility and modular typechecking. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 224–240. ACM Press, 2003.
15. Nathaniel Nystrom, Stephen Chong, et Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04 : Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 99–115. ACM Press, 2004.
16. Clemens A. Szyperski. Import is not inheritance — why we need both : Modules and classes. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, volume 615 sur *LNCS*, pages 19–32, Utrecht, the Netherlands, juin 1992. Springer-Verlag.
17. Allen Wirfs-Brock. Subsystems — proposal. OOPSLA 1996 Extending Smalltalk Workshop, octobre 1996.
18. Allen Wirfs-Brock et Brian Wilkerson. An overview of modular Smalltalk. In *Proceedings OOPSLA '88*, pages 123–134, novembre 1988.